AIAA-99-4456

PRACTICAL ISSUES IN ESTIMATING FAULT CONTENT AND LOCATION IN SOFTWARE SYSTEMS

Allen P. Nikora
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109-8099
Allen.P.Nikora@jpl.nasa.gov

Norman F. Schneidewind
Code IS/Ss
Naval Postgraduate School
Monterey, CA 93943
nschneid@nps.navy.mil

John C. Munson
Computer Science Department
University of Idaho
Moscow, ID 83844-1010
jmunson@cs.uidaho.edu

ABSTRACT

Over the past several years, techniques have been developed to discriminate between fault-prone software modules and those that are not, and to estimate a software system's residual fault content. These techniques can be applied during the stages of a development effort prior to test, thereby giving software managers greater visibility into the systems being developed and allowing them to exert more accurate and precise control into their quality. There are practical issues involved in implementing these measurement techniques in a production development environment. In this paper, we describe measurement techniques being implemented on a development effort at the Jet Propulsion Laboratory, identify implementation issues, and describe proposed resolutions to these issues.

INTRODUCTION

Over the past several years, techniques have been developed to use measurements of a software system's structure to discriminate between fault-prone software modules and those that are not, and to estimate a software system's residual fault content.[1, 2, 3, 4, 5] These techniques can be applied during the earlier stages of a development effort prior to test, thereby giving software managers greater visibility into their projects and allowing them to exert more accurate and precise control over the systems for which they are responsible.

Boolean discriminant functions (BDFs) and measures of Relative Critical Value Deviation (RCVD) can be used in classifying the quality of software during the quality control and prediction process.[4, 5] Using failure data from the Space Transportation System Primary Avionics Software System (STS PASS), these functions have been shown to provide good accuracy (i.e., 3% error) for classifying low quality software. This is true because the BDFs consist of more than just a set of metrics. They include additional information for discriminating quality: critical values. To form BDFs, nonparametric statistical methods are used to:

1. identify a set of candidate metrics for further analysis.
2. identify the critical values of the metrics. This computation is based on the Kolmogorov-Smirnov (K-S) test.
3. find the optimal BDF of metrics and critical values based on the ability of the BDF to satisfy *both* statistical (i.e., ability to classify quality) and application (i.e., quality achieved versus the cost to achieve it) criteria.

Detailed maps of a software system's residual fault content at any point in time can be constructed from its structural evolutionary and failure histories. We have shown that it is possible to identify a relationship between the measured amount of change between two successive versions of a software module and the number of faults inserted into that module, thereby providing an estimate of the rate of fault insertion.[2, 3, 6] This lets us estimate the number of faults inserted into each module of the system at any point during its development. The number of residual faults in each module is computed by subtracting the number of faults known to have been repaired in a module (taken from the system's failure history) from the estimated number of faults inserted into that system. Software managers can use this information to more accurately prioritize those modules to which fault identification and repair resources should be applied, thereby making the most effective use of limited resources.

Although our previous work has involved only the implementation phase, these methods can make use of software structural methods available prior to implementation, thereby allowing faulty modules to be identified early development phases. This is especially appealing since it has been repeatedly demonstrated that removing faults during the latter phases of a software development effort can be one or two orders of magnitudes more costly than removing those same faults during earlier development phases.[7] There are practical issues that must be addressed prior to implementing these methods on a software development effort. These involve:

1. Measuring workproducts such as specifications and designs, which are often expressed in a mixture of formal and informal notations, and may not be easily measurable.
2. Devising accurate, consistent, and practical methods of tracing discovered faults back to the point at which they originally inserted into the system. This is required in order to develop models relating the fault insertion rate to measurements of a system's structural evolution.

To resolve the first issue, we are currently investigating methods of translating the outputs of some of the more popular tools for diagramatically representing a system's behavior (i.e., statecharts) into forms that can easily be measured. With respect to the second issue, we are refining an ad-hoc taxonomy developed as part of our initial work and determining how it might be formalized. In the remainder of this paper, we briefly discuss the management techniques we are working to implement on the Mission Data System (MDS), a software development project at JPL which will produce the next generation of planetary exploration flight and ground software, discuss in more detail the practical issues associated with these techniques, and describe methods being considered for their resolution.

IDENTIFYING FAULT-PRONE SOFTWARE COMPONENTS

Measurements of a software system's structure can be used to discriminate between fault-prone modules and those that are not fault prone. During development, structural measurements of the system are taken and used to construct BDFs and the RCVD metrics. A BDF is a Boolean function consisting of AND and OR operators, module metric values, and metric critical values that is used to classify the quality of software.[5] A metric critical value is a value in the range of the metric, estimated by using the inverse of the Kolmogorov-Smirnov distance that provides a threshold between two levels (e.g., high and low) of the quality of the software.

In forming BDFs, it is important to perform a marginal analysis when making a decision about how many metrics to include. If many metrics are added at once to the set, the contribution of individual metrics is obscured. Also, the marginal analysis provides an effective rule for deciding when to stop adding metrics. If certain metrics are dominant in their effects on classifying quality, additional metrics are not needed to accurately classify quality. Related to this property of *dominance* is the property of *concordance*, which is the degree to which a set of metrics produces the same result in classifying software quality. A high value of *concordance* implies that additional metrics will not make a significant contribution to accurately classifying quality; hence, these metrics are redundant.

Note that the BDF provides only an accept/reject decision on a component's quality. The RCVD, which measures the extent to which a measurement deviates from its critical value, further indicates the extent to which a component's quality is above or below an acceptable level.

Taking the structural measurements necessary to form BDFs and compute the RCVD is a straightforward matter. Commercial measurement tools are readily available, and these can be easily integrated into modern configuration management tools, such as CCC Harvest or ClearCase, to make the measurements without requiring any extra effort on the part of the developers. In selecting the measurement tool and setting up the measurement process, the following decisions need to be made:

- At what point are the measurements to be made (e.g., at the completion of a specified build, at regular intervals)?
- What structural measurements will be taken? We have developed guideline for selecting appropriate measurements.[5] Whatever tool is selected should clearly identify how the measurements are taken.

The more difficult aspect is counting the number of problem reports associated with each component so that critical values for the BDFs may be computed. Our experience indicates that the most effective way of gathering this information is to choose a problem reporting system that integrates with the configuration management system. Links are established between each problem report and the components that are modified in response to that problem report. Relating problem reports to software components then becomes a simple matter of querying the problem report database.

We are currently working to implement such a measurement system for the MDS, the flight and ground software being developed at JPL for the next generation of planetary exploration spacecraft.

In principal, BDFs and RCVD can be extended to development activities prior to implementation. If structural information about specification or design artifacts is available, and if technical reviews such as Fagan inspections are regularly held to identify faults in the workproducts, then BDFs can be formed and RCVD values computed.[8] While there is an abundance of tools for measuring source code during the implementation phase, we have found it significantly more difficult to measure artifacts produced in earlier development phases. In many development efforts, we have observed that the syntax of the notations used in producing designs and specifications is not as well-defined as that of the source code, making it difficult to define a complete or consistent set of measurements. In many cases, designs and specifications are specified in a mixture of natural language and other informal or semiformal notations. This compounds the problem by introducing the possibility of incompatibilities between the notations.

To resolve the measurement problem, we are working in cooperation with the MDS to devise methods of measuring UML diagrams. Current MDS plans call for producing specifications and designs in the form of UML diagrams such as use cases, scenarios, class diagrams, and statecharts. Unfortunately, currently available UML tools do not provide structural measurements of the models they produce. However, there are ways of translating some of the output of these tools into forms that can be easily measured. For instance, the MOCES tool can be used to translate suitably-constrained statecharts produced by Statemate into the Promela modeling language used by the Spin model-checker.[9, 10] It is then a relatively straightforward matter to design and implement a structural analyzer to measure the statechart's Promela equivalent. We are investigating the practicality of doing this type of measurement in the MDS environment.

ESTIMATING FAULT INSERTION RATES AND FAULT CONTENT

Our previous work indicates that there is a strong relationship between measurements of a system's structural evolution and the rate at which faults are inserted during development.[3, 6] During implementation, these measurements can be taken at the level of individual modules (i.e., methods, functions). These measurements act as a fault surrogate – they are strongly related to the system's fault content, and they aggregate in the same manner as fault counts of individual modules aggregate into a total fault count for the system.

Using only measures of structural evolution, it is possible to estimate the proportional fault burden of a module at any point during its development. If the total amount of change that module *i* has undergone with respect to a baseline *B* is given by x_i^B, then its proportional fault burden d_i^B is given by:

$$d_i^B = x_i^B \Big/ \sum_{i=1}^{N} x_i^B \qquad (1)$$

where *N* is the number of modules comprising the system. This quantity can be compared to each module's proportion of faults discovered during test, g_i^B. Comparisons of d_i^B and g_i^B can be made to identify those modules to which additional fault identification resources should be allocated: If d_i^B is greater than g_i^B , then additional resources should be allocated to module *i*. Conversely, if g_i^B is greater than d_i^B , then too many resources have already been allocated to module *i*, and no further fault identification effort is required until d_i^B becomes equal or greater to d_i^B.

Once repaired, faults can be identified and traced back to the point at which they were first inserted into the system. This information can be used to construct a regression model relating the number of faults inserted per unit of structural change.[6] The number of faults inserted into the system at the module level can then be estimated, as can the residual fault content of each module. The modules with the largest estimated residual fault content can be identified, and allocated fault identification and repair resources proportional to their residual fault content.

The structural evolution of the software at the module level can be measured transparently to the developers. Modern configuration management tools make it straightforward matter to make structural measurements as part of checking a change package into the repository. These tools can be set up to start the measurement tool each time a developer checks a source file back into the repository, thereby making it unnecessary for the developer to perform any extra work related to measurement. The measurement history is automatically recorded, ready for the use by the test and quality assurance staff. We are currently working with MDS to integrate a measurement tool we have developed for this purpose, EMA, with the MDS configuration management system.

Tracing faults back to their point of origin is somewhat more complicated. Ideally, a problem report would identify each fault repaired in response to the reported failure, but we have found that this is not done for almost of the software development efforts we have studied. However, most development efforts require that the source code files that were changed in response to a reported failure be identified. It therefore becomes necessary to examine the changes made in response to a reported failure to identify the faults that have been repaired. We have developed a set of fault counting and identification rules to help analysts in this activity.[3] Once the faults have been identified, it is necessary to find the point at which they were first inserted into the source code, so that the relationship between the amount of structural change made to the system and the number of faults inserted can be established. This means searching all previous versions of a module prior to the point at which the fault was corrected. Unfortunately, this is largely a manual technique at this point. As part of our future work, we hope to formalize the fault counting and identification rules we have developed. We hope to develop a search tool which analysts could use to automate searches of all previous revisions of a module.

In identifying and counting faults, it is necessary to separate changes due to fault repair from those made due to change requests. We are developing a problem resolution process for the MDS that would give priority to fault repair. Before modifying or enhancing a component, developers should first repair known faults in a component and submit the repaired component to the repository. Developers would then check out the repaired component to which enhancements or other modifications could be made. In addition to reducing a source of noise in the measurements, it is a matter of good engineering practice to repair a component containing known faults before attempting to enhance it.

MEASURING TEST EFFICIENCY

Our recent work has also shown that test efficiency can be measured by comparing an ideal execution profile to the actual profile observed while executing the software during test.[1, 2, 6] The ideal execution profile is constructed from a detailed history of the software's structural evolution during its development. Since these measurements have been shown to be strongly related to the rate at which faults are inserted into the software system during its development, each module should then be tested according to the amount of change it has undergone since the last time it was tested. Suppose that a system about to enter test consists of a set of modules A, and that the cumulative amount of change that has been made to all the modules in A since the last round of testing is X. If a particular module a has incurred a total amount of change x since the last round of testing occurred, the proportion of time that should be spent executing module a is x/X. This ideal profile can then be quantitatively compared with the profile observed during test. The testing staff can then:

- Calculate a numerical value for the effectiveness of the test procedure(s).
- Identify those modules that were insufficiently tested, and the extent to which they were insufficiently tested.
- Identify those modules in which too much execution time was spent during test, and by how much the ideal execution time was exceeded.

The structural evolution of the source code is measured as described above. Measuring the execution profile may be somewhat more invasive, in that the software needs to be monitored during execution. The ideal way of accomplishing this is to design the necessary instrumentation into the software. However, this requires some effort to design and implement the software. Traditionally, software development efforts have not seen any benefit to devoting scarce resources to a capability that is not seen to directly affect the system's functionality. Another way of accomplishing this is to compile the software with instrumentation that will record the transitions from module to module during execution. Comparatively little effort is required to link the instrumentation package into the software system. However, the instrumentation may alter the software's behavior. This is particularly true in real time systems, for which changes in timing relationships may alter the system's behavior in unpredictable ways.

A third way of observing a system's execution profile is to build the necessary instrumentation into the testbed on which the system is run during the various testing phases. This is attractive for the following reasons:

- There is no need to link an instrumentation package into the software. The system's timing relationships will not be affected.
- Depending on the nature of the testbed, it may be possible to capture more extensive information about the system's behavior during execution than would be possible with either of the other two methods. There is a limit to how much information may be extracted from a system using built-in or linked-in instrumentation before its behavior starts to be adversely affected. However, if the instrumentation is implemented on the testbed, it is the testbed's rather than the system's performance that is affected. Even if the instrumentation places a relatively high load on the testbed, the system under test will behave in the same fashion as if the instrumentation placed a small load on the testbed.
- If the testbed is designed to be used for multiple missions, the instrumentation becomes part of the multimission capability. It is easier to use a capa-

bility that is already available rather than rebuilding it for each new mission.

We are working to implement this trace capability on the bit-level simulator that will be produced as part of the test environment for the MDS. We believe that only the following requirements need to be levied on the simulator to implement this capability:

- During execution, the simulator shall log to a user-specified file the following information each time control is passed from module to module:
    - The address to which control is transferred.
    - The time at which control is transferred. NOTE: Time may be expressed in terms other than seconds or fractions of seconds – it would be perfectly satisfactory to express elapsed time during a test as the cumulative number of (simulated) CPU clock cycles, for instance.
- Users shall be able to select whether or not they want to log an execution profile prior to starting a test.

Of course, there will be constraints on the size of the log file that can be produced, which will require decisions on the following matters:

- Should the user be allowed to specify the maximum size of the log file prior to a test run, and how should that size be specified (e.g., size in bytes, size in number of transitions, amount of execution time to be recorded)? What should be the simulator's response if the desired file size is too large?
- Should there be manual capabilities to start and stop logging?
- Once the log file becomes full, should logging stop, or continue at the beginning of the file? Should the user be given a choice?

There will also be issues relating to the performance of the simulator that will need to be addressed. Although a large instrumentation load on the simulator will not affect the behavior of the system under test, it may affect the amount of time that it takes to execute a test. In a real development effort, there will always be pressure to adhere to the schedule, and if the instrumentation produces delays in completing tests, there will be pressure to avoid using it.

Irrespective of the programming language chosen, we believe that this will be a relatively simple capability to implement in a bit-level simulator. This is because we will be working at low level of abstraction by trapping the small, well-defined set of instruction(s) that are used to transfer control between modules, rather than attempting to identify such transfers at a higher level of abstraction.

Finally, note that none of the analysis capabilities are to be implemented in the simulator. If the log file is available to members of the test and quality assurance staff, analysis of the results can be done in non-real time.

ESTIMATING RISK OF EXPOSURE TO RESIDUAL FAULTS

Once estimates of residual fault content have been made at the module level, this information can be combined with dynamic information obtained during test and fielded use to estimate the system's risk of exposure to residual faults.[3] The practical issues involved here are the same as those of computing test efficiency, namely obtaining measurements of the system's structural evolution and its execution profile during test and fielded use. As previously noted, measuring the system's structural evolution can be done as part of the configuration management process, transparently to the development team. For MDS, we intend to build into the bit-level simulator on which the flight components will be tested the capability of measuring and recording the execution profile of the system under test.

MEASURING REQUIREMENTS RISK

One of the software maintenance problems of any development organization is to evaluate the risk of implementing requirements changes. These changes can affect the reliability and maintainability of the software. As part of our work on MDS, we are determining the applicability of the risk assessment used on the STS flight software. To assess the risk of change, the software development contractor uses a number of risk factors, which are described below. The risk factors were identified by agreement between NASA and the development contractor based on assumptions about the risk involved in making changes to the software. This formal process is called a risk assessment. No requirements change is approved by the change control board without an accompanying risk assessment. During risk assessment, the development contractor attempts to answer such questions as: "Is this change highly complex relative to other software changes that have been made on the Shuttle?" If this is the case, a high-risk value would be assigned for the complexity criterion. To date this *qualitative* risk assessment has proven useful for identifying possible risky requirements changes or, conversely, providing assurance that there are no unacceptable risks in making a change. However, there has been no *quantitative* evaluation to determine whether, for example, high risk factor software was really less reliable and maintainable than low risk factor software. In addition, there is no model for predicting the reliability and maintainability of the software, if the change is implemented. The intent of our work with MDS is to address both of these issues.

We had considered using requirements attributes like completeness, consistency, correctness, etc., as risk factors.[11] While these are useful generic concepts, they are difficult to quantify. Although some of the following risk factors also have qualitative values assigned, there are a number of quantitative factors, and many of the factors deal with the execution behavior of the software (i.e., reliability), which is our research interest.

The following are the definitions of the risk factors, where we have placed the factors into categories and have provided our interpretation of the question the factor is designed to answer. In addition, we added the risk factor *requirements specifications techniques* because we feel that this one could represent the highest reliability risk of all the factors if a technique leads to misunderstanding of the intent of the requirements. If the answer to a yes/no question is "yes", it means this is a high-risk change with respect to the given factor. If the answer to a question that requires an estimate is an anomalous value, it means this is a high-risk change with respect to the given factor.

Complexity Factors

- Qualitative assessment of complexity of change (e.g., very complex)
  - Is this change highly complex relative to other software changes that have been made on the Shuttle?
- Number of modifications or iterations on the proposed change
  - How many times must the change be modified or presented to the Change Control Board (CCB) before it is approved?

Size Factors

- Number of lines of code affected by the change
  - How many lines of code must be changed to implement the change?
- Size of data and code areas affected by the change
  - How many bytes of existing data and code are affected by the change?

Criticality of Change Factors

- Whether the software change is on a nominal or off-nominal program path (i.e., exception condition)
  - Will a change to an off-nominal program path affect the reliability of the software?
- Operational phases affected (e.g., ascent, orbit, and landing)
  - Will a change to a critical phase of the mission (e.g., ascent and landing) affect the reliability of the software?

Locality of Change Factors

- The area of the program affected (i.e., critical area such as code for a mission abort sequence)
  - Will the change affect an area of the code that is critical to mission success?
- Recent changes to the code in the area affected by the requirements change
  - Will successive changes to the code in one area lead to non-maintainable code?
- New or existing code that is affected
  - Will a change to new code (i.e., a change on top of a change) lead to non-maintainable code?
- New or existing code that is affected
  - Will the change be on a path where only a small number of system or hardware failures would have to occur before the changed code is executed?

Requirements Issues and Function Factors

- Number and types of other requirements affected by the given requirement change (requirements issues)
  - Are there other requirements that are going to be affected by this change? If so, these requirements will have to be resolved before implementing the given requirement.
- Possible conflicts among requirements changes (requirements issues)
  - Will this change conflict with other requirements changes (e.g., lead to conflicting operational scenarios)
- Number of principal software functions affected by the change
  - How many major software functions will have to be changed to make the given change?

Performance Factors

- Amount of memory required to implement the change
  - *Will the change use memory to the extent that other functions will be not have sufficient memory to operate effectively?*
- Effect on CPU performance
  - *Will the change use CPU cycles to the extent that other functions will not have sufficient CPU capacity to operate effectively?*

Personnel Resources Factors

- Number of inspections required to approve the change.
- Manpower requirements required to implement the change
  - *Will the manpower required to implement the software change be significant?*

Manpower required to verify and validate the correctness of the change

- *Will the manpower required to verify and validate the software change be significant?*

Tools Factor

- Any software tools creation or modification required to implement the change
  - *Will the implementation of the change require the development and testing of new tools?*
- Requirements specifications techniques (e.g., flow diagram, state chart, pseudo code, control diagram).
  - *Will the requirements specification method be difficult to understand and translate into code?*

We have access to several sets of data from the Space Shuttle of the following types:

A. Pre-release and post release failure data from the Space Shuttle from 1983 to the present.
B. Risk factors for the Shuttle *Three Engine Out Auto Contingency* and *Single Global Positioning System* software. This software was released to NASA by the developer on 10/18/95 and 3/5/97, respectively.
C. Metrics data for 1400 Shuttle modules, each with 26 metrics.

We will use the Shuttle data to test our hypothesis about the ability of risk factors to discriminate between levels of reliability and complexity, and will work with the MDS to apply our findings. We will also attempt to identify more quantitative risk factors than those given above, based on structural measurements of the UML specifications and designs that will be produced by the MDS. This project provides a rare opportunity to work with the software development team and testers to establish a measurement plan form the inception of a project as opposed to the usual situation of having to intervene in an on-going project. We plan to instrument the software system for obtaining measurements throughout the development and maintenance process.

CONCLUSION

This research is another in the series of our software measurement projects that has included software reliability modeling and prediction, metrics analysis, risk analysis, and maintenance stability analysis.[2, 3, 4, 12] We have been involved in the development and application of software reliability models for many years.[13, 14] Our models, as is the case in general in software reliability, use failure data as the driver. This approach has the advantage of using a metric that represents the dynamic behavior of the software. However, this data is not available until the test phase. Predictions at this phase are useful but it would be much more useful to predict at an earlier phase – preferably during requirements analysis—when the cost of error correction is relatively low. Thus, there is great interest in the software reliability and metrics field in using static attributes of software in reliability modeling and prediction.

Our earlier work indicates that structural measurements of a software system are strongly related to the system's fault content. The techniques to analyze this information and produce the desired estimates are based on well-understood multivariate analysis techniques, and are appropriate performed by members of a software quality assurance team. The structural measurements required to make these estimates can be easily collected as part of the development process, requiring little or no additional effort on the part of the development staff. Estimates of proportional fault content can be made by using structural information alone. We are currently working with the MDS project at JPL to implement these measurement mechanisms.

The failure and fault information required to calibrate the models, on the other hand, may be more difficult to collect. Modern configuration management and problem reporting systems can automate much of the data collection process – failure reports can be associated with the files and modules that were changed in response to a failure report. Although this will require developers to provide that information on problem report forms as part of problem identification and repair activity, this does not represent any additional effort in that developers are already required to provide this information.

Identification and counting of the faults, however, is still largely a manual process, as is tracing them back to their points of origin. Although this is work that need not be performed by the development team, resources do need to be allocated to the quality assurance staff members that will perform this type of analysis. Our experience indicates that for a project the size of MDS, roughly one or two additional analysts would have to be added to a development team several dozen strong. As part of our future work, we hope to develop automated techniques for assisting analysts in this area.

ACKNOWLEDGMENTS

The research described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology. Portions of the work were sponsored by the National Aeronautics and Space Administration's IV&V Facility
.

REFERENCES

1 J. C. Munson and G. A. Hall, "Estimating Test Effectiveness with Dynamic Complexity Measurement," Empirical Software Engineering Journal. Feb. 1997

2 J. Munson, A. Nikora, "Estimating Rates of Fault Insertion and Test Effectiveness in Software Systems", proceedings of the Fourth ISSAT International Conference on Quality and Reliability in Design, Seattle, WA, August 12-14, 1998

3 A. Nikora, N. Schneidewind, J. Munson, "IV&V Issues in Achieving High Reliability and Safety in Critical Control System Software," JPL D-15740, final report, January 19, 1998

- Volume 1 – "Measuring and Evaluating the Software Maintenance Process and Metrics-Based Quality Control"
- Volume 2 – "Measuring Defect Insertion Rates and Risk of Exposure to Residual Defects in Evolving Software Systems"
- Volume 3 – "Appendices"

4 Norman F. Schneidewind, "Reliability Modeling for Safety Critical Software", IEEE Transactions on Reliability, March 1997

5 Norman F. Schneidewind, "Predicting Deviations in Software Quality by Using Critical Value Deviation Metrics, to be published in the proceedings of the Tenth International Symposium on Software Reliability Engineering, Boca Raton, FL, November 1-4, 1999.

6 A. Nikora, J. Munson, "Determining Fault Insertion Rates for Evolving Software Systems", proceedings of the Ninth International Symposium on Software Reliability Engineering, Paderborn, Germany, November 4-7, 1998

7 B. W. Boehm, Software Engineering Economics, Prentice-Hall, Inc., 1981

8 M. E. Fagan, "Design and Code Inspections to Reduce Errors in Program Development," IBM Systems Journal, Volume 15, Number 3, pp 182-211, 1976

9 E. Mikk, Y. Lakhnech, and M. Siegel, "Towards Efficient Modelchecking Statecharts: A Statecharts to Promela Compiler," In 3rd International SPIN Workshop, University of Twente, April 97

10 G. Holzmann, "The Model Checker Spin," IEEE Trans. on Software Engineering, Vol. 23, No. 5, May 1997, pp. 279-295

11 Alan Davis, Software Requirements: Analysis and Specifications, Prentice-Hall, Englewood Cliffs, NJ, 1990

12 Norman F. Schneidewind, "How to Evaluate Legacy System Maintenance", IEEE Software, Vol. 15, No. 4, July/August 1998, pp. 34-42. Also translated into Japanese and reprinted in: Nikkei Computer Books, Nikkei Business Publications, Inc., 2-1-1 Hirakawacho, Chiyoda-Ku, Tokyo 102 Japan, 1998, pp. 232-240

13 Norman F. Schneidewind, "Software Reliability Model with Optimal Selection of Failure Data", IEEE Transactions on Software Engineering, Vol. 19, No. 11, November 1993, pp. 1095-1104

14 Norman F. Schneidewind and T. W. Keller, "Application of Reliability Models to the Space Shuttle", IEEE Software, Vol. 9, No. 4, July 1992 pp. 28-33